Technical Project I

Richard Ogujawa (L00145067)

December 2, 2023

Title: Technical Project I

Supervisor: Professor Shagufta Henna

Degree: MSc Data Science

1 Executive Summary

1.1 Introduction

Accompanying the increase of the number of vehicles on the road (Nealon, 2023), came an increase in the number of road fatalities. According to a statement by Statement by Jonathan Adkins, CEO of Governors Highway Safety Association (GHSA): "Traffic deaths have surged 30% over the past decade, with nearly 10,000 more fatalities when compared to 32,893 in 2013." Innovations are being made towards mitigating this, such as smart systems and head-up displays in cars, however the issue persists. The goal of this project was to contribute to the concerted efforts made towards making the roads safer.

1.2 Objectives

The primary objectives of this project was to produce a Machine Learning (ML) model that would be able to make predictions regarding the likelihood of a severe crash given a set of parameters like time, date and location, to help caution drivers and encourage safer driving.

1.3 Findings

The dataset provided didn't have any underlying structure upon which a reliable model could be generated to solve this particular problem.

2 Dataset Description

In order to conduct this project, the 'Motor Vehicle Collisions Crashes' table was used. It contains over 2.04 million rows of data and 29 columns. Sourced from NYC police reports the dataset comprises observations summarising the details of a large number of road crashes which occurred in New York City dating back to July 2012. The dataset is regularly updated, as such, it also includes the most recent motor vehicle collisions.

3 Methods

3.1 Stage 1: Loading the Data

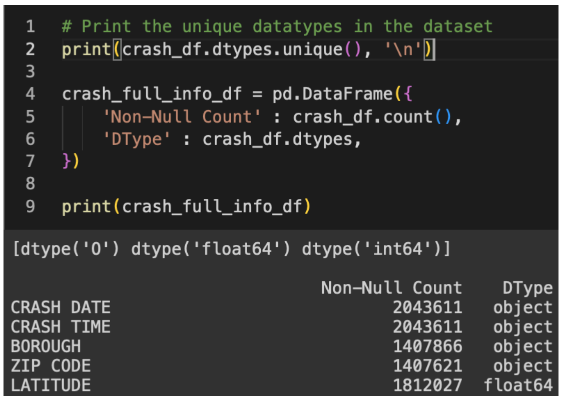
Due to the large size of the data the dataset was uploaded to Google Drive, and subsequently retrieved using the `drive.mount()` static method from the google.colab module.

3.2 Understanding The Data

Once loaded, the metadata was inspected to provide a deeper understanding of the data being analysed. This provided some insights into, for example, what data cleaning/transformation processes needed to be done before the dataset could progress to the Feature Engineering stage. The following was gleaned during the inspection of the metadata:

- **Some columns stored strings:** Models tend to work better with numeric data, so the presence of these string values meant that some method needed to be applied to transform these occurrences of non-numeric data into numeric data before progressing.

- **There are null values present:** This meant that some method(s) needed to be applied to appropriately handle the null values present in the dataset.



```
# Print the unique datatypes in the dataset
print(crash_df.dtypes.unique(), '\n')

crash_full_info_df = pd.DataFrame({
    'Non-Null Count' : crash_df.count(),
    'DType' : crash_df.dtypes,
})

print(crash_full_info_df)
```

```
[dtype('O') dtype('float64') dtype('int64')]

                              Non-Null Count    DType
CRASH DATE                           2043611   object
CRASH TIME                           2043611   object
BOROUGH                              1407866   object
ZIP CODE                             1407621   object
LATITUDE                             1812027  float64
```

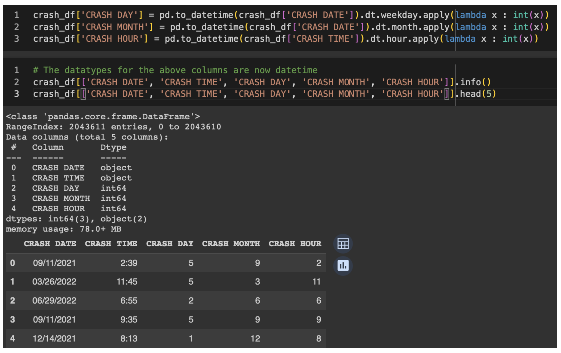
Figure 1: MetaData Description for Dataset

3.3 Stage 2: Data Cleaning and Transformation

3.3.1 Converting String Dates to Datetime Data Type

According to the metadata provided so far, two of the columns which were expected to become features - 'CRASH DATE' and 'CRASH TIME' - were stored as strings. However, for use in this project, they were converted to a more appropriate datatype - datetime.

Also, rather than using specific crash dates and times, they were converted into crash day, month and hour to allow for more generic use. For example, instead of making predictions based on specific dates or times, predictions could now be generalised to a certain day of the week, month, or hour of a particular crash.



```
crash_df['CRASH DAY'] = pd.to_datetime(crash_df['CRASH DATE']).dt.weekday.apply(lambda x : int(x))
crash_df['CRASH MONTH'] = pd.to_datetime(crash_df['CRASH DATE']).dt.month.apply(lambda x : int(x))
crash_df['CRASH HOUR'] = pd.to_datetime(crash_df['CRASH TIME']).dt.hour.apply(lambda x : int(x))
```

```
# The datatypes for the above columns are now datetime
crash_df[['CRASH DATE', 'CRASH TIME', 'CRASH DAY', 'CRASH MONTH', 'CRASH HOUR']].info()
crash_df[['CRASH DATE', 'CRASH TIME', 'CRASH DAY', 'CRASH MONTH', 'CRASH HOUR']].head(5)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2043611 entries, 0 to 2043610
Data columns (total 5 columns):
 #   Column       Dtype
---  ------       -----
 0   CRASH DATE   object
 1   CRASH TIME   object
 2   CRASH DAY    int64
 3   CRASH MONTH  int64
 4   CRASH HOUR   int64
dtypes: int64(3), object(2)
memory usage: 78.0+ MB
```

	CRASH DATE	CRASH TIME	CRASH DAY	CRASH MONTH	CRASH HOUR
0	09/11/2021	2:39	5	9	2
1	03/26/2022	11:45	5	3	11
2	06/29/2022	6:55	2	6	6
3	09/11/2021	9:35	5	9	9
4	12/14/2021	8:13	1	12	8

Figure 2: Converting Strings Dates to Datetime Data Type

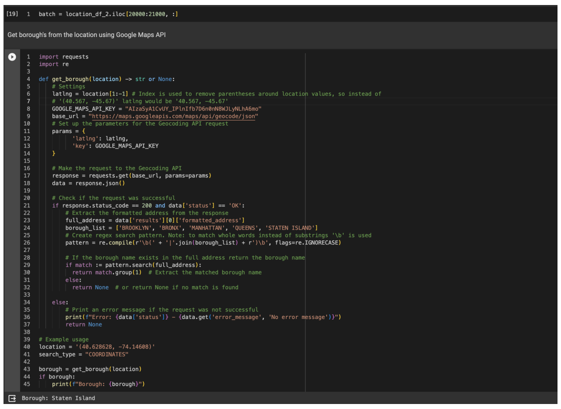
3.3.2 Handling Null Values

After rectifying this, there still existed the issue of the null values present in the dataset. Missing data impede upon ML algorithms and their ability to affirm assumptions upon which they can generate suitable models.

The decided-upon method implemented to handle this issue was data enrichment. The only attribute this was done for was the 'BOROUGH' column (which specified the borough that the crash occurred in). The reason why this specific location-related attribute was chosen was because there are only 5 boroughs in New York, which can easily be converted into numeric data. Similar to the reasoning for extracting hours, months and days from 'CRASH DATE' and 'CRASH TIME', the focus on boroughs as opposed to exact locations, would allow for more broadly-scoped analyses.

For the places where borough name were missing, requests were made to the Google Maps API using the longitude and latitude values present in the row to obtain the full address that belonged to that coorindate. The corresponding borough was then extracted from that address using regular expressions. Given that the dataset is so big, this data enrichment process was done in batches using 1,000 rows at a time.

After the data had been processed it was stored in a table in SQLite.



```
[19]  batch = location_df_2.iloc[20000:21000, :]
```

Get borough's from the location using Google Maps API

```python
import requests
import re

def get_borough(location) -> str or None:
    # Settings
    latlng = location[1:-1] # Index is used to remove parentheses around location values, so instead of
    # '(40.567, -45.67)' latlng would be '40.567, -45.67'
    GOOGLE_MAPS_API_KEY = "AIzaSyA1CvUY_IPlnIfb7D6n0nN8WJLyNLhA6mo"
    base_url = "https://maps.googleapis.com/maps/api/geocode/json"
    # Set up the parameters for the Geocoding API request
    params = {
        'latlng': latlng,
        'key': GOOGLE_MAPS_API_KEY
    }

    # Make the request to the Geocoding API
    response = requests.get(base_url, params=params)
    data = response.json()

    # Check if the request was successful
    if response.status_code == 200 and data['status'] == 'OK':
        # Extract the formatted address from the response
        full_address = data['results'][0]['formatted_address']
        borough_list = ['BROOKLYN', 'BRONX', 'MANHATTAN', 'QUEENS', 'STATEN ISLAND']
        # Create regex search pattern. Note: to match whole words instead of substrings '\b' is used
        pattern = re.compile(r'\b(' + '|'.join(borough_list) + r')\b', flags=re.IGNORECASE)

        # If the borough name exists in the full address return the borough name
        if match := pattern.search(full_address):
            return match.group(1)  # Extract the matched borough name
        else:
            return None  # or return None if no match is found

    else:
        # Print an error message if the request was not successful
        print(f"Error: {data['status']} - {data.get('error_message', 'No error message')}")
        return None

# Example usage
location = '(40.628628, -74.14608)'
search_type = "COORDINATES"

borough = get_borough(location)
if borough:
    print(f"Borough: {borough}")
```

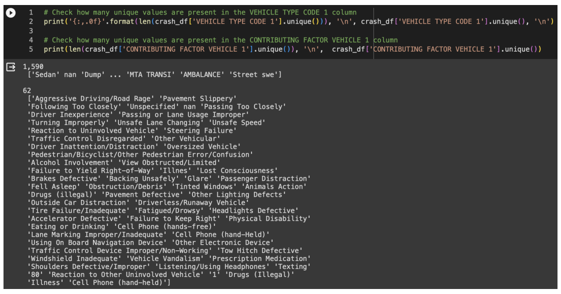
```
Borough: Staten Island
```

Figure 3: Data Enrichment using Google Maps API PT.I

The newly formed dataframe and the original one were merged using a left outer join on the collision_id column. This ensured that all of the data from the original dataframe was included in the merged dataframe. Finally only the columns deemed to be essential for the project were kept in the dataframe and null values were dropped.

In the research proposal, the column 'VEHICLE TYPE CODE 1' was suggested as a possible feature for the model. However, upon further investigation it was apparent that it would be difficult to use given that there were over 1,590 unique values in it. To allow for this column, as well as the other columns that share a like name ('VEHICLE TYPE CODE 2', 'VEHICLE TYPE CODE 3', etc.) to be used they would first need to be placed in categories which generalise the data values, and provide the possibility for one-hot encoding. However, the values present in this column range from clear and concise vehicle descriptions such as 'Sedan' or 'Taxi' to more obscure entries, such as 'sanit' or 'OMT'. Trying to guess the categories of the more obscure values would potentially produce inaccurate training data for the Machine learning algorithm, resulting in the production of an inaccurate model.

The 'CONTRIBUTING FACTOR VEHICLE' columns were omitted for a similar reason.



```python
# Check how many unique values are present in the VEHICLE TYPE CODE 1 column
print('{:,.0f}'.format(len(crash_df['VEHICLE TYPE CODE 1'].unique())), '\n', crash_df['VEHICLE TYPE CODE 1'].unique(), '\n')

# Check how many unique values are present in the CONTRIBUTING FACTOR VEHICLE 1 column
print(len(crash_df['CONTRIBUTING FACTOR VEHICLE 1'].unique()), '\n',  crash_df['CONTRIBUTING FACTOR VEHICLE 1'].unique())
```

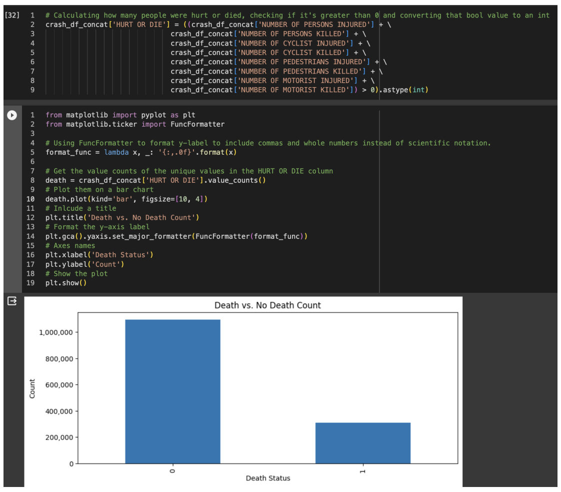
```
1,590
 ['Sedan' nan 'Dump' ... 'MTA TRANSI' 'AMBALANCE' 'Street swe']

62
 ['Aggressive Driving/Road Rage' 'Pavement Slippery'
 'Following Too Closely' 'Unspecified' nan 'Passing Too Closely'
 'Driver Inexperience' 'Passing or Lane Usage Improper'
 'Turning Improperly' 'Unsafe Lane Changing' 'Unsafe Speed'
 'Reaction to Uninvolved Vehicle' 'Steering Failure'
 'Traffic Control Disregarded' 'Other Vehicular'
 'Driver Inattention/Distraction' 'Oversized Vehicle'
 'Pedestrian/Bicyclist/Other Pedestrian Error/Confusion'
 'Alcohol Involvement' 'View Obstructed/Limited'
 'Failure to Yield Right-of-Way' 'Illnes' 'Lost Consciousness'
 'Brakes Defective' 'Backing Unsafely' 'Glare' 'Passenger Distraction'
 'Fell Asleep' 'Obstruction/Debris' 'Tinted Windows' 'Animals Action'
 'Drugs (illegal)' 'Pavement Defective' 'Other Lighting Defects'
 'Outside Car Distraction' 'Driverless/Runaway Vehicle'
 'Tire Failure/Inadequate' 'Fatigued/Drowsy' 'Headlights Defective'
 'Accelerator Defective' 'Failure to Keep Right' 'Physical Disability'
 'Eating or Drinking' 'Cell Phone (hands-free)'
 'Lane Marking Improper/Inadequate' 'Cell Phone (hand-Held)'
 'Using On Board Navigation Device' 'Other Electronic Device'
 'Traffic Control Device Improper/Non-Working' 'Tow Hitch Defective'
 'Windshield Inadequate' 'Vehicle Vandalism' 'Prescription Medication'
 'Shoulders Defective/Improper' 'Listening/Using Headphones' 'Texting'
 '80' 'Reaction to Other Uninvolved Vehicle' '1' 'Drugs (Illegal)'
 'Illness' 'Cell Phone (hand-held)']
```

Figure 4: Number of Unique Values for 'VEHICLE TYPE CODE 1' Column

3.3.3 Turning Categorical Data into Numeric Data

As was aforementioned, the 'BOROUGH' column was going to be used in the predictive model and ,therefore, had to be transformed into some form of numeric data. Given that it was nominal categorical data with more than two classes, one-hot encoding was implemented instead of label-encoding.

The final part of the data transformation process was to check for class imbalances and resolve them if present. In order to do this, the label was generated for the dataframe, which was simply a summation of 8 other columns which together gave the total of how many people were injured or died as a result of a given crash. Then the code checked if this value was greater than 0, if so then the crash was a severe crash and it was given a True value, and if not then it was given a False value. These boolean values are then converted to the integers 1 and 0 respectively.



```
[32]
# Calculating how many people were hurt or died, checking if it's greater than 0 and converting that bool value to an int
crash_df_concat['HURT OR DIE'] = ((crash_df_concat['NUMBER OF PERSONS INJURED'] + \
                                  crash_df_concat['NUMBER OF PERSONS KILLED'] + \
                                  crash_df_concat['NUMBER OF CYCLIST INJURED'] + \
                                  crash_df_concat['NUMBER OF CYCLIST KILLED'] + \
                                  crash_df_concat['NUMBER OF PEDESTRIANS INJURED'] + \
                                  crash_df_concat['NUMBER OF PEDESTRIANS KILLED'] + \
                                  crash_df_concat['NUMBER OF MOTORIST INJURED'] + \
                                  crash_df_concat['NUMBER OF MOTORIST KILLED']) > 0).astype(int)
```

```
from matplotlib import pyplot as plt
from matplotlib.ticker import FuncFormatter

# Using FuncFormatter to format y-label to include commas and whole numbers instead of scientific notation.
format_func = lambda x, _: '{:,.0f}'.format(x)

# Get the value counts of the unique values in the HURT OR DIE column
death = crash_df_concat['HURT OR DIE'].value_counts()
# Plot them on a bar chart
death.plot(kind='bar', figsize=[10, 4])
# Inlcude a title
plt.title('Death vs. No Death Count')
# Format the y-axis label
plt.gca().yaxis.set_major_formatter(FuncFormatter(format_func))
# Axes names
plt.xlabel('Death Status')
plt.ylabel('Count')
# Show the plot
plt.show()
```

Figure 5: Checking Class Imbalances

3.4 Stage 3: Feature Engineering

It was evident that not all the features would be useful to the training of the model as such, only the important ones were kept. After selecting the features, the data was investigated to reveal any patterns, relationships or correlations that would help provide some intuition about how the predictions would likely turn out. An interesting relationship to consider was the correlation between the features, as well as the correlation between each feature and the label. A few graphs and plots were created in Tableau to help visualise these relationships.

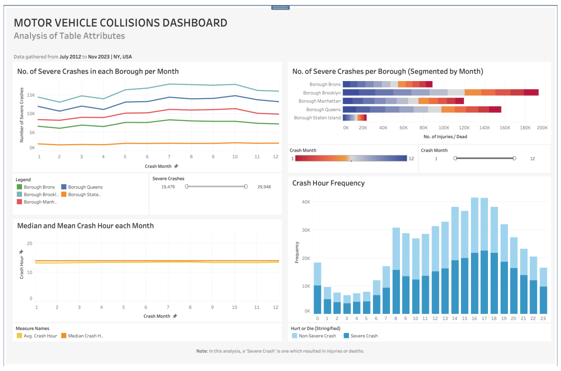


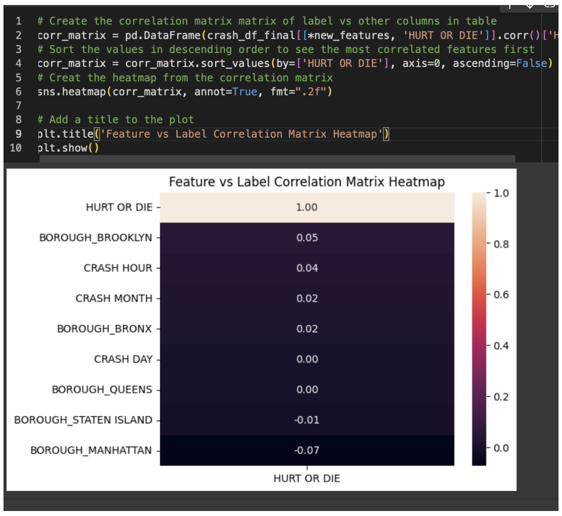
Figure 6: Graphs and Plots in Tableau Dashboard

3.4.1 Feature Analysis

A correlation matrix was also generate to deduce how much each feature is affected by another. Based on the heatmap that the correlation matrix generated it's evident that there are no strong inter-columnar relationships present in the dataset.

The strongest relationships that existed between the columns were the relationships between the borough-related columns which made sense because one-hot encoding was used to generate these columns. Therefore, the presence of a 1.0 value in one column necessitated the presence of a 0.0 value in all of the other borough-related columns.

The correlation between the features and the label was also plotted and visualised using a heatmap. It was then sorted from the strongest correlation to the weakest to identify which columns had the most effect on the label. It was evident that all of the features had little to no effect on the label.

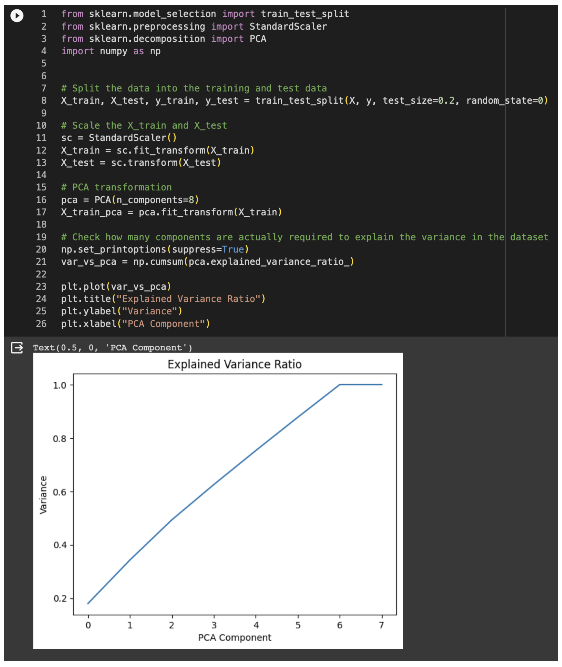


```
# Create the correlation matrix matrix of label vs other columns in table
corr_matrix = pd.DataFrame(crash_df_final[[*new_features, 'HURT OR DIE']].corr()['H
# Sort the values in descending order to see the most correlated features first
corr_matrix = corr_matrix.sort_values(by=['HURT OR DIE'], axis=0, ascending=False)
# Creat the heatmap from the correlation matrix
sns.heatmap(corr_matrix, annot=True, fmt=".2f")

# Add a title to the plot
plt.title('Feature vs Label Correlation Matrix Heatmap')
plt.show()
```

Figure 7: Heatmap Showing Correlation Between Features and Label

3.4.2 Principal Component Analysis(PCA)

PCA was used to improve the quality of the features to aid in the production of an reliable model. To ensure an accurate result from the PCA algorithm, the data is normalised to the same scale before PCA is applied.

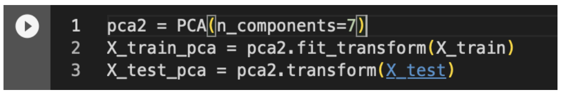


```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import numpy as np


# Split the data into the training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Scale the X_train and X_test
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# PCA transformation
pca = PCA(n_components=8)
X_train_pca = pca.fit_transform(X_train)

# Check how many components are actually required to explain the variance in the dataset
np.set_printoptions(suppress=True)
var_vs_pca = np.cumsum(pca.explained_variance_ratio_)

plt.plot(var_vs_pca)
plt.title("Explained Variance Ratio")
plt.ylabel("Variance")
plt.xlabel("PCA Component")
```

```
Text(0.5, 0, 'PCA Component')
```

Figure 8: Applying PCA

After graphing the Explained Variance Ratio, it was clear that only seven components were required to explain 100% of the variance for the dataset, given that the knee point of the curve was at the seventh component.



```
pca2 = PCA(n_components=7)
X_train_pca = pca2.fit_transform(X_train)
X_test_pca = pca2.transform(X_test)
```

Figure 9: PCA was Re-Applied Using Optimal Number of Components

3.5 Stage 4: Model Selection

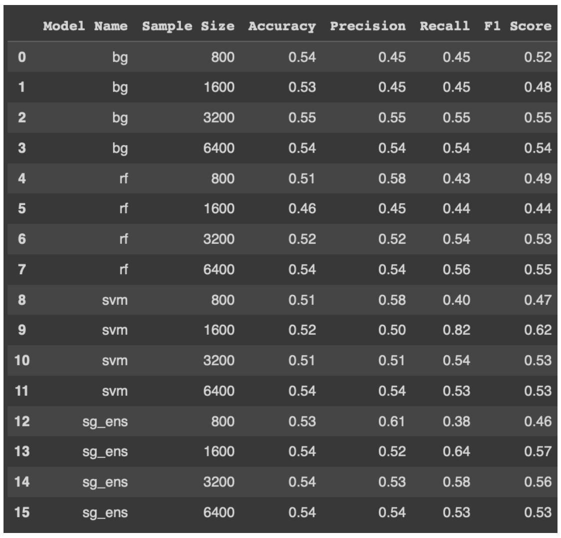
As was expressed in the project proposal, the Bagging Estimator AND Random Forest algorithms were the algorithms chosen for this project. They were chosen due to the fact that they both provided classifiers and also had measures in place to reduce variance (the models sensitivity to noise in the training data). These measures therefore resulted in models less prone to over-fitting, and more likely to produce more accurate generalisations, which is highly important given the severity of the problem that the model is being designed to solve.

Added to this, an Support Vector Machine (SVM) algorithm and a Stacking Classifier algorithm were also implemented with the goal of producing more accurate models after failing to do so using the previously mentioned classifiers. These two models were picked because: (1) SVMs tend to be effective at capturing complex patterns in data, which the other two estimators seemed to be struggling to do. (2) The Stacking Classifier is an ensemble method and as such was chosen to combine all of the estimators and hopefully average better predictions.

GridSearchCV was also implemented to find the best hyper-parameters for each model. A handful of parameters were chosen and tweaked over time to arrive at an optimal set of parameters given the dataset. This took the models from being simple traditional models to being more complex and better-suited to the specific situation it was being created for.

4 Conclusion and Results

Despite applying PCA, converting non-numeric data to numeric data, trying various sample sizes and testing out a number of hyper-parametric configurations the key performance metrics (accuracy score, recall, precision and f1 score) remained around 0.5. This suggested that the dataset lacked the underlying structure (or set of relationships) necessary to produce a model which would make accurate predictions given the initial set of hypotheses.



	Model Name	Sample Size	Accuracy	Precision	Recall	F1 Score
0	bg	800	0.54	0.45	0.45	0.52
1	bg	1600	0.53	0.45	0.45	0.48
2	bg	3200	0.55	0.55	0.55	0.55
3	bg	6400	0.54	0.54	0.54	0.54
4	rf	800	0.51	0.58	0.43	0.49
5	rf	1600	0.46	0.45	0.44	0.44
6	rf	3200	0.52	0.52	0.54	0.53
7	rf	6400	0.54	0.54	0.56	0.55
8	svm	800	0.51	0.58	0.40	0.47
9	svm	1600	0.52	0.50	0.82	0.62
10	svm	3200	0.51	0.51	0.54	0.53
11	svm	6400	0.54	0.54	0.53	0.53
12	sg_ens	800	0.53	0.61	0.38	0.46
13	sg_ens	1600	0.54	0.52	0.64	0.57
14	sg_ens	3200	0.54	0.53	0.58	0.56
15	sg_ens	6400	0.54	0.54	0.53	0.53

Figure 10: Performance of the Different Models at Different Sample Sizes

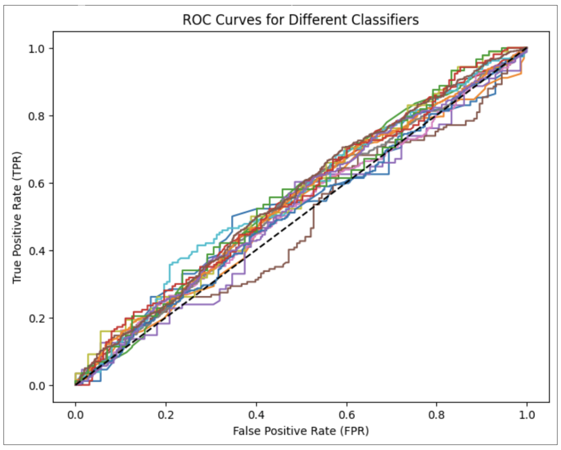


Figure 11: ROC Curve for the Performance of the Different Models at Different Sample Sizes

The failure of this project to produce a suitable model highlights the importance of feature analysis and a deeper understanding of the dataset prior to committing it to a entire analytical pipeline, which will be a primary focus going forward.

References

- Nealon. (2023). 2023 Car ownership statistics. [Online] Available at: https://www.bankrate.com/insurance/car/car-ownership-statistics/ [Accessed 2 December 2023].

- GHSA. (2023). 2023 Small Decrease in 2022 Traffic Deaths Sustains Pandemic-Fueled Surge... [Online] Available at: https://www.ghsa.org/resources/news-releases/NHTSA-2022-Traffic-Deaths23 [Accessed 2 December 2023].